
SHORTER IS NOT CHEAPER: A CONTROLLED COMPARISON OF DECLARATIVE ANIMATION SPECIFICATIONS AND FREE-FORM CODE FOR LLM-GENERATED EXPLANATIONS

A PREPRINT

Jiantang Huang

huang.jiant@northeastern.edu

October 26, 2026

ABSTRACT

Chat assistants increasingly answer conceptual questions with interactive, animated explanations, generated either as free-form HTML/JavaScript or as declarative specifications that a fixed runtime plays. The declarative route is widely expected to be cheaper, faster and more reliable, because its outputs are shorter and constrained. We test this expectation in controlled comparisons whose hypotheses and analysis plans were frozen before the runs. We designed *ExplainSpec*, a time-indexed declarative animation format, and an observation contract that makes every generated explanation programmatically inspectable; validated, pacing-agnostic checkers score 20 explanation requests from five domains. Three Claude models (360 jobs) and, in a follow-up, the open-weight Qwen3-14B with thinking enabled and disabled (240 jobs) answered every request in both formats. For every model the specifications were 25–48% shorter in answer tokens (output excluding reasoning), yet with reasoning enabled total output fell significantly only for the largest Claude model and no model generated significantly faster. The reasons differed. The Claude models reasoned far more before writing specifications (exploratory; for the two larger models, from 15–20% to 85–88% of generations). Qwen3-14B reasoned about as much in both formats (a pre-registered test of this shift was not supported), and its shorter answers saved little against a reasoning trace more than twice as long. Nor were specifications more reliable: free-form pages were valid after one repair round in 95–100% of jobs for every model, whereas Qwen3-14B produced valid specifications in only 33% of jobs and passed all semantic tests in 17% (HTML/JavaScript: 62%). For reasoning models, a shorter format alone therefore does not make generation cheaper, faster or more reliable, and a format learned in context can cost reliability.

Keywords Generative UI · Declarative animation · LLM output representation · Latency · Reasoning tokens

1 Introduction

Interactive, visual answers have become a standard capability of chat assistants: deployed systems explain mathematics and science with parameterised visuals and simulations inside the conversation [1–4]. Because such explanations are generated on demand, while the user waits, their generation cost and latency matter as much as their quality. Two ways of producing them coexist. The model can write a complete, self-contained HTML/JavaScript page, which is maximally expressive but long: complete generated pages can take a minute or more [5]. Alternatively, the model can emit a compact declarative description that a trusted, prebuilt runtime renders, the approach taken by generative-UI catalogs such as A2UI, json-render and OpenUI [6–8] and, according to its announcement, by ChatGPT’s Intelligent UI [4].

The case for the declarative route rests on two intuitions. First, because decoding output tokens dominates generation latency [9], a shorter artifact should arrive sooner and cost less. Second, a constrained format with a validating runtime should fail less often than free-form code. Neither intuition has been tested under controlled conditions for animated explanations, and the indirect evidence points in different directions: a structured trace pipeline for algorithm videos improved correctness but was reported to consume more tokens than direct code [10]; a harness that generates

component-constrained UI code reported fewer output tokens than raw HTML without controlling for quality [11]; format restrictions can interact with reasoning [12–14]; and frontier models already produce running interactive pages most of the time, although often with wrong content [15].

Testing these intuitions fairly is harder than it appears. The two formats must convey the same content, so that differences can be attributed to the representation rather than to the task; correctness must be checked for animated, parameterised output regardless of how the model paces its animation; and the generating model must be isolated from the experimenter’s tools and instructions. We meet these requirements as follows. Both formats implement one programmatic observation contract, so that the same checkers apply to both. The checkers compare state traces with ground truth independently of pacing, and they were validated against independent reference implementations and by mutation testing before any model output existed. Every generation runs in an isolated process, and the hypotheses, measures and analysis plan were written down and frozen before the main run. Within this design, three Claude models of different sizes answered 20 requests in physics, algorithms, network protocols, machine learning and probability, three times in each format, with one repair round for invalid artifacts. In a follow-up with its own frozen plan, we repeated the study with an open-weight reasoning model from another family, Qwen3-14B, run locally with thinking enabled and, as an ablation, disabled.

The results do not support either intuition for the models we tested (fig. 2, section 6). With reasoning enabled, the specifications are 25–48% shorter in answer tokens for all four models, yet generation is not significantly faster for any of them, and total output falls significantly only for the largest Claude model. Why the saving vanishes differs between the families: the Claude models reason far more before writing a specification, whereas Qwen3-14B, for which a pre-registered test of this shift was not supported, reasons at length in both formats, so that its much shorter answer changes the total little. Nor is the constrained format more reliable. Free-form pages are valid after at most one repair round in 95–100% of jobs for every model, while specifications are less often valid; for Qwen3-14B they pass all semantic tests in only 17% of jobs, against 62% for HTML/JavaScript. The central lesson is that output compression and generation savings are different quantities: what a reasoning model writes is only part of what it generates, and a format learned in context can make the explanation itself less reliable.

Contributions. (1) Evidence from controlled comparisons with four models from two families that a declarative animation format shortens the answer but, with reasoning enabled, neither speeds up generation nor improves reliability, and that the reasons differ between the model families (sections 5 and 6). (2) ExplainSpec, a time-indexed declarative animation format with runtime-provided playback and parameter controls, together with an observation contract that makes free-form and declarative explanations equally inspectable (section 3). (3) A benchmark of 20 explanation tasks with pacing-agnostic checkers, validated by 40 independent reference implementations and mutation testing, and a taxonomy of the errors models make in an in-context-learned declarative language (sections 4 and 5).

2 Related Work

Generative UI: free-form code versus declarative catalogs. Interactive answers are now produced in two ways. Systems that let the model write complete HTML/CSS/JavaScript achieve high quality, but a page can take a minute or more to generate, and the pipeline relies on post-processors to fix common errors [5]; Gemini’s generative interfaces follow this route. Deployed assistants also offer in-chat visuals and simulations [2, 3]; according to its documentation, Claude builds them with HTML [16], whereas the format of the Gemini app’s simulations is not publicly documented. The second route constrains the model to a catalog of declarative components that a trusted client renders, as in A2UI, json-render and OpenUI [6–8], and, according to its announcement, in ChatGPT’s Intelligent UI, whose representation is not public [4]. Three observations motivate a controlled test. First, the efficiency evidence for compact UI formats is indirect: a harness that generates component-constrained UI code reported fewer output tokens than raw HTML without controlling for quality [11, Sec. 3.1.1]. Second, small models struggle with a catalog they have not been trained on: base models with 2B and 4B parameters produce valid A2UI for only 3% and 62% of requests, whereas a fine-tuned 4B model approaches the quality of a frontier model at a fraction of its cost [17]. Third, existing catalogs describe interface widgets and playback of pre-rendered media [18], and json-render’s video renderer turns JSON timelines of clips into video compositions [7]; none combines time-indexed animation with user-adjustable parameters from which the runtime recomputes it, which is what interactive conceptual explanations require.

LLM-generated explanatory animations. Systems that do generate explanatory animation mostly have the model write code: agents write Manim programs for educational videos and spend 18–26 times the tokens of direct code generation [19], and the InteractScience benchmark finds that self-contained HTML/JavaScript demonstrations often run while their science is wrong [15]. Several lines come closer to a declarative design. Keyframer lets a model emit CSS keyframes that the browser plays [20]. DataMagic’s multi-agent pipeline writes a declarative data-video

specification that executes more reliably than direct code, although it took about three times as long [21], and spec-then-compile pipelines with specification repair have been applied to interactive what-if interfaces, whose direct one-shot generation often fails to render [22]. ALGOGEN is the closest comparison: the model writes a tracker program whose schema-checked traces drive fixed renderers, which made algorithm videos more correct but, as reported, consumed more tokens than direct code [10]. These systems produce offline video, standalone animations or static interfaces, and none measures generation latency inside a conversation. The vocabulary itself is not new either: interpreted animation command streams date back to algorithm animation [23], and visualisation grammars already compose timed transitions [24, 25]. Our contribution is therefore not the vocabulary but the controlled comparison of what it costs a model to emit such a specification instead of code.

Output format, reasoning and latency. Because decoding dominates latency, shorter outputs should be faster [9]. Whether a format changes how a model reasons is less clear. Restricting answers to structured formats can lower reasoning accuracy [12], but a vendor analysis attributes much of this effect to prompt and parser artifacts [13], and it concentrates in models near their capacity [14], which also reports how many tokens a non-reasoning model writes under different formats; conversely, code is a more effective action format than JSON when steps must be composed [26]. For reasoning models answering benchmark questions, the number of thinking tokens tracks question difficulty rather than whether the answer must follow a structured format [27]. To our knowledge, no study has measured how the representation of a generated artifact changes this hidden reasoning, and with it latency and cost, or compared a time-indexed declarative animation format with free-form code on identical tasks while measuring artifact tokens, reasoning tokens, latency and correctness. This is the gap our study addresses.

3 ExplainSpec and the observation contract

Format. An ExplainSpec document is a JSON object whose main parts are (fig. 1) `params` (user-adjustable parameters with ranges), `constants`, `state` (initial values), `steps` (the timeline), `derived` (values recomputed every frame), `expose` (the variables reported to an inspector) and `objects` (what is drawn), plus an optional title and canvas size. The timeline is a sequence of step instances. Entering an instance applies literal assignments (`set`) and then expressions (`compute`); the new values hold for `dur` seconds. Steps can be repeated (`repeat`), looped while a condition holds (`while`) or grouped, so that the runtime can compute a whole algorithm trace from the parameters. The timeline variables `time`, `step` (counted from 0) and the in-step progress `p` allow continuous motion, for example binding a physical time to `p` and positions to closed-form expressions. Seven object types are available: `rect`, `circle`, `line`, `text`, `path` (a curve sampled from expressions), `array` (a row of cells with highlighting) and `repeat` (instantiating child objects for an index range). Any object property can be a literal or an expression introduced by `=`.

Expression language. Expressions look like JavaScript but are parsed and interpreted by the runtime instead of being executed as code. They support numbers, strings, arrays, object literals, indexing, member access, arithmetic, comparison, logical and conditional operators, and a whitelist of pure functions (`sin`, `floor`, `clamp`, `lerp`, `len`, `push`, `swap`, `setAt`, `slice`, `contains`, ...). There are no statements, loops inside expressions, user-defined functions or global objects; iteration is expressed with `repeat` and `while` steps. The validator checks every expression for syntax errors and unknown variables before playback, and the timeline is capped at 2,000 step instances.

Runtime. The runtime (about 600 lines of JavaScript) validates the document, compiles the timeline for the current parameter values, renders the scene as SVG, and provides play/pause, a scrubber, step buttons and one slider per parameter. Changing a parameter recompiles the timeline.

Observation contract. To evaluate free-form and declarative artifacts with the same checkers, every artifact must expose `window.__explain` with a duration, the `params` with their current values, a deterministic `seek(t)`, a `getState()` returning the task’s state variables at the current time, and `setParam(name, value)`. The ExplainSpec runtime implements the contract automatically from `expose`, deriving the exposed state and the drawing from the same variables; a free-form page has to implement the contract itself. Each distinct discrete state must remain visible for at least 0.3 s, so that dense sampling cannot miss it.

4 Experimental design

The design implements the three requirements stated in the introduction: the tasks and prompts give both formats the same content, the evaluation checks correctness independently of pacing and was validated before use, and every generation is isolated from the experimenter’s environment.

```
{ "params": { "a0": { "min": 1, "max": 200, "default": 84, "step": 1 },
              "b0": { "min": 1, "max": 200, "default": 36, "step": 1 } },
  "state": { "done": false },
  "steps": [
    { "dur": 1.5, "compute": { "a": "a0", "b": "b0", "done": "b0 == 0" } },
    { "while": "b != 0", "dur": 1.5,
      "compute": { "r": "a % b", "a": "b", "b": "r", "done": "r == 0" } },
    { "dur": 2, "label": "= \"b = 0, so gcd = \" + a" } ],
  "expose": [ "a", "b", "done" ],
  "objects": [
    { "type": "rect", "x": 100, "y": 150, "w": "=min(600, a * 3)", "h": 40,
      "text": "'a = ' + a" },
    { "type": "rect", "x": 100, "y": 220, "w": "=min(600, b * 3)", "h": 40,
      "text": "'b = ' + b", "visible": "=b > 0" } ] }
```

Figure 1: An abridged ExplainSpec document (one of the two worked examples given to the models): Euclid’s algorithm, computed by the runtime from the parameters with a while step.

Table 1: The 20 tasks. Each task is tested at its default and 1–2 further parameter settings (“Settings”).

Domain	Task	Parameters	Required state variables	Settings
Physics	Projectile motion	v0, theta	t, x, y	3
Physics	Simple pendulum	L, theta0	t, angle	3
Physics	Damped spring-mass oscillator	k, c	t, x	3
Physics	RC circuit charging	R, C	t, v_c, i	3
Algorithms	Binary search	target	lo, hi, mid, found	3
Algorithms	Quicksort partition (Lomuto)	variant	arr, j, pivot	3
Algorithms	Breadth-first search	start	current, queue, discovered	3
Algorithms	Hash table resizing	variant	capacity, size, buckets	3
Protocols	TCP three-way handshake	client_isn, server_isn	client_state, server_state, last_segment	3
Protocols	TCP slow start and congestion avoidance	ssthresh0, loss_round	round, cwnd, ssthresh	3
Protocols	DNS iterative resolution	tld_cached	msg, answered	2
Protocols	Go-Back-N sliding window	window, lost	round, base, sent, delivered	3
ML	Gradient descent	lr, w0	iter, w, loss	3
ML	Backpropagation through one neuron	x, w, b, target	phase, z, y, L, dL_dy, dL_dz, dL_dw, dL_db	3
ML	k-means clustering	init	iter, centroids, assign	3
ML	Scaled dot-product attention	q1	stage, scores, scaled, weights, output	3
Prob/math	Monty Hall problem	pick	car, opened, stay_wins, switch_wins	3
Prob/math	Distribution of the sum of dice	n	k, dist	3
Prob/math	Fourier series of a square wave	terms	k, value_at_pi_2, amplitude	3
Prob/math	Bayesian updating of a coin	a0, b0	n, alpha, beta, posterior_mean	3

Tasks. We wrote 20 explanation requests, four per domain (table 1). Each request states the question, the parameters the user can change (range, default, step), the state variables the explanation must expose (names, units, meanings) and precise rules wherever the subject admits several conventions, for example the Lomuto partition scheme, the tie-breaking rule of k -means or the congestion-window update of TCP. Every task is tested at 2–3 parameter settings chosen so that an explanation that ignores the parameter fails. Content that changes over time is the substance of every task, the condition under which animation is most plausibly helpful, although the evidence that animation aids learning more than static graphics is not encouraging [28].

Conditions and prompts. In the *HTML/JS* condition the model writes one self-contained page (no network access) that animates the explanation, offers controls and implements the observation contract. In the *Spec* condition the model writes an ExplainSpec document. Both prompts contain the identical task brief, the same rules (names and units, 0.3 s per discrete state, consistency of drawing and state) and the same two worked examples, for two development tasks that are not part of the evaluation (uniform circular motion and Euclid’s algorithm), each written in the condition’s own format. The HTML/JS examples therefore contain code for the contract and the controls that the model can adapt, whereas the Spec examples rely on the runtime for both. The HTML/JS prompt additionally contains the contract and the Spec prompt the format documentation, so that the prompts have similar lengths (for the TCP handshake task, 12.3k characters for HTML/JS and 11.2k for Spec). Prompts were never revised in response to outputs of the models under study. While the reference implementations of the evaluation tasks were being written (see *Validation of the checkers* below), however, the expression language was extended (object literals and member access, without which the TCP handshake task could not be expressed) and the format documentation was clarified; the documentation was then checked on two smoke-test tasks that are not evaluated either. If an artifact is invalid, the model receives one repair

prompt with the errors a developer would see (load errors, uncaught exceptions, console errors, runtime or validator errors, contract problems and blocked network requests), never the semantic test results; this feedback has the same format in both conditions.

Models and generation. We used three models of different sizes through the Claude Code command-line interface (version 2.1.293): claude-opus-5-5, claude-sonnet-5-5 and claude-haiku-5-5, all with effort low. Every generation ran in a separate headless process with a neutral system prompt (plus a short fixed preamble that the interface inserts, identical in every call), no tools, no Model Context Protocol (MCP) servers, no user settings or plugins beyond the interface’s three built-in ones, no persistent session and an empty working directory. We verified the isolation without relying on the model’s self-report: the interface’s initialisation record lists no tools and no servers, and in a positive control with the user-level instruction file enabled, the same probe call is 989 tokens longer and the model quotes that file. Each of the 20 tasks $\times$ 2 conditions $\times$ 3 models was generated three times; we call one such generation, together with its possible repair, a *job* (360 jobs, that is, 180 pairs). Jobs ran in an order shuffled within each repetition, so that the conditions were interleaved in time, with four jobs running concurrently.

Evaluation. A Playwright harness loads every artifact in headless Chrome (fixed viewport, network blocked) and records the first contentful paint and the time at which the contract becomes callable. The harness reads the state only through the contract; it does not operate the visible controls. An artifact is *valid* if it loads without an uncaught exception and the contract is complete. For every tested parameter setting the harness calls `setParam`, then samples `getState()` every 0.05 s over the whole timeline (at most 6,000 samples), and a task-specific checker compares the trace with a ground-truth model. Checkers are pacing-agnostic: for discrete processes they compare the sequence of distinct states (and required intermediate configurations, such as the queue after each breadth-first-search expansion) with the ground truth; for continuous processes they check the physical relationships at every sample (e.g. $y = v_0 \sin \theta t - gt^2/2$ within 1% of the maximum height plus 1 cm) and the coverage of the required time span. A job *passes* if the final artifact is valid and all tested settings pass. Two descriptive checks complement the tests: a determinism probe (the same time gives the same state regardless of the order of `seek` calls) and a scene-change check (the rendered pixels differ between 20% and 80% of the timeline, with the form controls hidden). Generation timing is taken from the interface’s timestamped event stream, measured from its session-initialisation event: time to first token (first content delta), time to first answer token (first text delta), and generation latency (until the final result). Time to first render adds the first contentful paint, and time to first working interaction adds the time until the contract is callable.

Validation of the checkers. Before any model output existed, five independently instructed AI coding agents (Claude-based sub-agents of the coding assistant used to build the harness) that saw only the task briefs, the contract and the format documentation, but not the checker code, wrote one reference implementation per task in each format. All 40 pass. We then corrupted every reference trace systematically (parameter ignored, a variable perturbed, a discrete value dropped, two values swapped, truncation, a variable removed, a frozen animation). Of 1,430 non-trivial mutants, 1,408 (98.5%) are rejected; all 22 survivors were inspected and are equivalent with respect to the task rules (for example, dropping an intermediate state that the rules do not require).

Pre-registration and analysis. Hypotheses, measures, exclusion rules and the analysis plan were written down, and all files defining the experiment (prompts, examples, documentation, tasks, checkers, runtime, harness, generator and orchestrator) were frozen with SHA-256 hashes before the first main-run generation; the hashes were unchanged after the run. The pre-registration was internal rather than lodged with an external registry, and the analysis script was not among the hashed files. **H1:** the Spec condition needs fewer output tokens and less generation time (one-sided). **H2:** the Spec condition is valid at least as often, with a larger advantage for the smaller model (one-sided). **H3:** the semantic pass rates differ (two-sided). **H4** (descriptive): which tasks the Spec condition can express. The unit of analysis is the task: for each model we average (rates) or take the median (tokens, latency) over repetitions and analyse the 20 paired task-level differences with the Wilcoxon signed-rank test [29], using its exact permutation distribution, bootstrap 95% confidence intervals (CIs) over tasks (10,000 resamples) [30], and Holm’s correction across the three models within each measure [31]. A smoke test on two non-evaluated tasks, run before the freeze, showed that the models reason even at effort low, and more for specifications; we therefore added answer tokens (output minus reasoning tokens) as a secondary measure before the freeze. All other analyses of reasoning are exploratory. Generation cost US\$14.00 for the main run and US\$14.77 including the smoke test and the post-hoc repairs of section 5.4, as estimated by the interface at list prices.

Table 2: Outcomes by model and format (60 jobs each). Token and latency columns are medians over first attempts (latency is generation time); answer tokens exclude reasoning tokens, which the interface reports as “thinking tokens”; reasoning jobs: first attempts with any reasoning (of 60). Cost is the total for the 60 jobs, for first attempts only and including repair rounds.

Model	Format	Valid (%)		All tests pass (%)		Output tokens (median)			Reasoning jobs	Latency (s)	Cost (\$, 60 jobs)	
		1st	repair	1st	repair	total	answer	reasoning			1st	repair
Opus	HTML/JS	100	100	100	100	2,806	2,795	0	9	21.1	4.43	4.43
	Spec	88	100	83	95	2,472	2,043	402	51	20.7	3.89	4.63
Sonnet	HTML/JS	100	100	95	95	2,595	2,558	0	12	13.6	2.13	2.13
	Spec	95	100	85	90	2,886	1,822	933	53	18.4	2.16	2.32
Haiku	HTML/JS	100	100	93	93	5,538	2,926	2,454	56	21.6	0.20	0.20
	Spec	85	97	72	83	6,504	1,706	4,710	60	28.0	0.26	0.30

Table 3: Pre-registered tests on 20 paired tasks. Brackets: 95% bootstrap CIs. H1 and the secondary measure use first attempts, H2 and H3 the results after repair. Exact Wilcoxon signed-rank tests, one-sided in the hypothesised direction for H1, H2 and the secondary measure and two-sided for H3, with Holm adjustment across the three models; “–”: all differences are zero, so no test is possible; pp: percentage points.

Measure	Effect	Opus	Sonnet	Haiku
H1: total output tokens	median ratio Spec/HTML Holm-adjusted p	0.81 [0.76, 0.93] 0.040	0.99 [0.89, 1.16] 1.000	1.13 [0.96, 1.70] 1.000
H1: generation latency	median ratio Spec/HTML Holm-adjusted p	0.90 [0.80, 1.01] 0.368	1.23 [0.96, 1.37] 1.000	1.25 [1.04, 1.87] 1.000
H2: valid (with repair)	mean difference (pp) Holm-adjusted p	+0 [+0, +0] –	+0 [+0, +0] –	–3 [–8, +0] 1.000
H3: all tests pass (with repair)	mean difference (pp) Holm-adjusted p	–5 [–10, +0] 0.750	–5 [–18, +7] 0.750	–10 [–23, +0] 0.750
Secondary: answer tokens	median ratio Spec/HTML Holm-adjusted p	0.70 [0.64, 0.81] <0.001	0.75 [0.57, 0.85] <0.001	0.60 [0.51, 0.67] <0.001

5 Results

This section reports the main study with the three Claude models; section 6 reports the follow-up. All 360 jobs completed without infrastructure failure, and no sampling timed out. Table 2 summarises the outcomes and table 3 the pre-registered tests. We first ask whether the shorter format makes generation cheaper and faster (section 5.1) and, because it does not, where the saved tokens go (section 5.2). We then test the second intuition, that a constrained format is more reliable (section 5.3), check that these conclusions survive a correction and relaxed scoring (section 5.4), and finally examine which errors the declarative language itself induces (section 5.5).

5.1 The artifact shrinks, but generation does not get cheaper

Specifications are much shorter than HTML pages: the median artifact has 3,581 characters against 5,868, and answer tokens fall to 0.60–0.75 of the HTML/JS value for every model (fig. 2, left; all Holm-adjusted $p < 0.001$). Total output tokens, however, fall only for Opus (0.81, Holm $p = 0.040$ within the measure and 0.080 when all six H1 tests are adjusted together); for Sonnet they are unchanged (0.99), and for Haiku the point estimate is 1.13 with a CI that includes 1, with a heavy tail of long specification generations (90th percentile 14,557 vs 8,702). Generation latency follows total output (fig. 5): it is not significantly reduced for any model (Opus 0.90, CI [0.80, 1.01]); for Sonnet the point estimate is 1.23 with a CI that includes 1, and for Haiku it is 1.25 with a CI that excludes 1 in the direction opposite to the hypothesis (a reverse test was not pre-registered). Rendering adds little (median first contentful paint 32 ms vs 40–44 ms, the latter including the inlined runtime), so the median time to first render of first-attempt-valid artifacts follows generation latency (HTML/JS vs Spec: 21.1 vs 19.8 s for Opus, 13.7 vs 18.5 s for Sonnet, 21.6 vs 28.2 s for Haiku); including the repair rounds changes these medians by at most 1 s, and the contract becomes callable within 0.1 s of first render. On first attempts the Spec condition is cheaper for Opus (US\$3.89 vs 4.43 for 60 jobs) but not for Sonnet (2.16 vs 2.13) or Haiku (0.26 vs 0.20); including the repair rounds it is more expensive for all three models (table 2). H1 is supported only for the total tokens of the largest model.

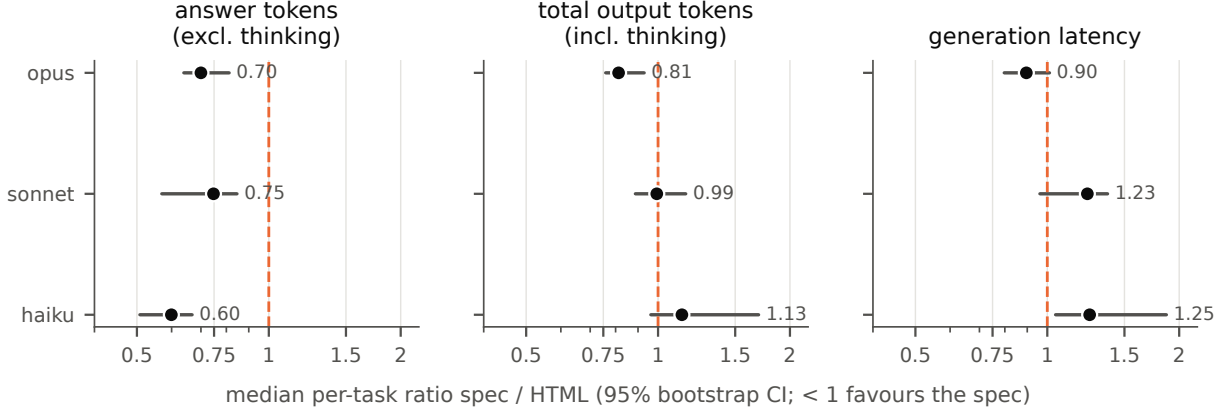


Figure 2: Median per-task ratio Spec/HTML/JS with 95% bootstrap CIs (log scale; dashed line: no difference). Specifications are consistently shorter in answer tokens; total output, which includes reasoning, falls only for Opus, and generation latency is not significantly reduced for any model.

5.2 Where the savings go: reasoning before writing

The difference between answer and total tokens is reasoning (table 2 and fig. 6); the analyses in this subsection are exploratory. With effort low, Opus and Sonnet reasoned in only 9 and 12 of 60 HTML/JS generations but in 51 and 53 of 60 Spec generations; Haiku reasoned in almost every generation and roughly doubled its median reasoning in the Spec condition (2,454 to 4,710 tokens). The shift is consistent across tasks: specifications drew more reasoning in 17 of 20 tasks for Opus and 15 of 20 for Sonnet, and less in only one task (exact signed-rank test, $p < 0.001$ for both). It is strongest for algorithm tasks, where the share of generations with reasoning rises from 0% to 100% for both models, and weakest for physics tasks (from 25–42% to 67%). The reasoning is also visible in the timing: the median time to the first *answer* token is 0.9 vs 5.3 s for Opus, 0.9 vs 9.3 s for Sonnet and 12.4 vs 22.9 s for Haiku, whereas the median time to the first token of any kind is 0.7–1.6 s in both conditions. For Opus, the added reasoning offsets part of the saving (a median of about 350 additional reasoning tokens against 875 fewer answer tokens per task); for Sonnet and Haiku it exceeds the saving (870 vs 680 and 2,350 vs 1,180 tokens). Section 7 examines two mechanisms that can produce this shift.

5.3 Free-form pages were always valid; specifications were not more correct

The second intuition holds that a constrained format fails less often than free-form code. Free-form code, however, left little room for improvement: every one of the 180 HTML/JS artifacts was valid on the first attempt, without a single uncaught exception. Specifications were valid on the first attempt in 85–95% of cases; 19 Spec jobs (7 Opus, 3 Sonnet, 9 Haiku) and no HTML/JS job needed the repair round, after which specifications were valid in 100% (Opus, Sonnet) and 97% (Haiku) of cases (fig. 3). The Spec condition was thus equally often valid after repair for the two larger models and less often for Haiku, and less often valid on the first attempt for all three; H2, including its prediction of a larger advantage for the smallest model, is not supported. After repair, all semantic tests were passed by 93–100% of HTML/JS jobs and 83–95% of Spec jobs. The differences (−5, −5 and −10 points for Opus, Sonnet and Haiku) have the same direction for all models but are not significant (exact Holm-adjusted $p = 0.75$ for each). Because 14–17 of the 20 tasks are at ceiling in both conditions, few tasks differ: for Opus and Haiku only three, so that the smallest attainable two-sided p is 0.25 (0.75 after Holm adjustment); for Sonnet six tasks differ, but in both directions ($p = 0.625$). H3 is therefore inconclusive. On the first attempt the difference is larger: in an exploratory count, 173 of 180 HTML/JS jobs and 144 of 180 Spec jobs pass, and all 12 tasks whose pass rate differs when pooled over models favour HTML/JS. Every task was solved by at least one specification (H4), although the language had been extended before the freeze so that the TCP handshake task could be expressed (section 4). Descriptively, the rendered scene of first-attempt-valid artifacts differed between the 20% and 80% points of the timeline in 97% (HTML/JS) and 98% (Spec) of cases, and every one of them passed the determinism probe.

To see what lies behind these differences, table 4 classifies the 26 jobs that did not pass, each under its first matching category. Sixteen are classified as strict readings of the briefs: eight Spec jobs add a leading state with index 0 where the brief starts at 1, which the timeline variable `step`, counted from 0, may induce; three HTML/JS jobs reset `mid` to null between binary-search probes; and five fall under an ambiguity in the Go-Back-N brief that a reference-implementation

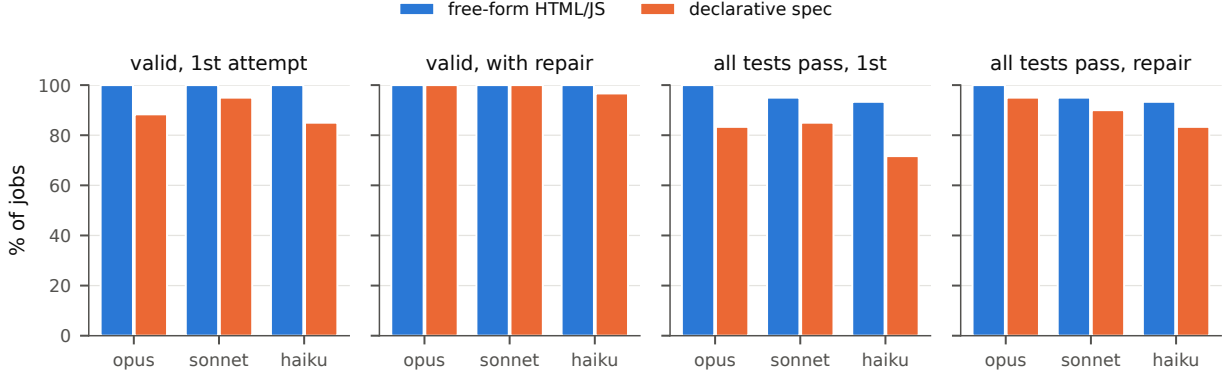


Figure 3: Validity and semantic pass rates by model and format, on the first attempt and after the single repair round (frozen protocol).

Table 4: Classification of the 26 jobs that do not pass all semantic tests after repair (frozen protocol). Each job is counted in the first matching category. In brackets: jobs that still fail after the correction and with relaxed checkers (section 5.4), that is, jobs with errors beyond the strict readings and the harness asymmetry.

Category	HTML/JS	Spec
Wrong content (wrong algorithm step, early stop, wrong update)	1 (1)	4 (4)
Still invalid after the repair round	0 (0)	2 (2)
Extra leading state with index 0 (brief: index starts at 1)	0 (0)	8 (2)
Binary search: <code>mid = null</code> between probes (brief: only before the first)	3 (0)	0 (0)
Go-Back-N: <code>sent</code> grows within a round (ambiguity in the brief)	3 (0)	2 (2)
Harness asymmetry (runtime error swallowed, no repair; section 5.4)	0 (0)	3 (0)
Total (of 180 jobs each)	7 (1)	19 (10)

agent had reported before the run (we kept the frozen interpretation). Three are affected by the harness asymmetry discussed next. Counting only errors beyond these readings and the asymmetry, one HTML/JS job and ten Spec jobs fail, for example through an incorrectly implemented partition step or a breadth-first search that stops after two nodes. Per-task results are shown in fig. 4 and per-task tokens in fig. 6 (section A).

5.4 The conclusions survive a post-hoc correction and relaxed scoring

Because several of these failures depend on how strictly the briefs are read, and some on how the harness treats runtime errors, we checked whether either affects the conclusions. A post-run audit found that the ExplainSpec runtime catches errors raised while a frame is computed or drawn, whereas an HTML/JS page throws the equivalent error out of `seek()` and is therefore invalid and repaired. Five Spec artifacts for the DNS task, which indexed an object with a dynamic key, were thus scored valid without repair (three with null state, two with undrawn message arrows). We keep the frozen results as primary and, in a documented deviation, re-evaluated all 379 artifacts with a corrected copy of the harness that treats such errors like exceptions: every other artifact reproduced its frozen outcome (validity, passed tests and, for first attempts, failure messages), confirming that the harness is deterministic, and the five passed after repair. An exploratory analysis additionally relaxed the three strict readings for both formats. Neither the correction nor the relaxation changes the direction of any result (table 5); under relaxed scoring the Spec condition is within one or two jobs of HTML/JS for Opus and Sonnet but remains 12 points less often correct for Haiku.

5.5 Errors in an in-context-learned declarative language

The lower first-attempt validity of specifications has a visible source in the errors that the validator and the runtime report: of all Spec artifacts, 26 triggered such an error (table 6). The most frequent errors carry JavaScript habits into a language that only looks like JavaScript: indexing an object with a dynamic key (`px[msg.from]`), calling array functions on a state that was initialised to null, or placing the object-property prefix = inside fields that are always expressions. Free-form JavaScript, which the models have seen far more often, produced no uncaught exception at all.

Table 5: Jobs passing all semantic tests after repair (%, HTML/JS vs Spec) under the frozen protocol, after treating runtime errors like exceptions (corrected), and additionally with relaxed checkers (exploratory).

Analysis	Opus	Sonnet	Haiku
Primary (frozen protocol)	100 vs 95	95 vs 90	93 vs 83
Runtime errors treated as exceptions (corrected)	100 vs 97	95 vs 93	93 vs 83
Corrected, with relaxed checkers (exploratory)	100 vs 98	98 vs 97	100 vs 88

Table 6: Validator and runtime errors in Spec artifacts (all attempts, first error per artifact).

Error raised by the runtime or validator	Artifacts
Object indexed with a dynamic key / member access on a non-object	8
Array function applied to a non-array (e.g. a null initial state)	4
= prefix inside a plain expression field (<code>dur</code> , <code>derived</code>)	4
Malformed expression (unbalanced brackets, unquoted text)	3
Non-terminating <code>while</code> loop	2
Derived variable used inside a step	2
Other (invalid JSON, JavaScript spread syntax, non-string expression)	3
Total spec artifacts with an error (all attempts)	26

6 Follow-up with an open-weight reasoning model

Design. To test whether the results depend on the model family, and to run the condition that the Claude command-line interface did not allow, reasoning disabled, we repeated the study with Qwen3-14B [32], an open-weight model whose thinking can be enabled or disabled per request. The model (14.8B parameters, 4-bit quantisation Q4_K_M) was served locally by Ollama 0.34.3 on one consumer GPU with 16 GB of memory, entirely on the GPU, with a context of 32,768 tokens and at most 24,576 generated tokens per call. Tasks, prompts, worked examples, format documentation, contract, checkers, artifact extraction and repair round were the frozen files of the main study; the generator sent one request per attempt with the same neutral system prompt, and the corrected harness of section 5.4 scored the artifacts from the start. Each mode, thinking enabled and disabled, covered 20 tasks $\times$ 2 formats $\times$ 3 repetitions (240 jobs in total), generated one at a time with the sampling settings that the model’s documentation recommends for each mode and a fixed seed per attempt. Reasoning tokens were counted from the streamed thinking chunks, whose number differed from the server’s token count by less than 1%, and latency was measured from the request, with the model already loaded; it includes prompt processing, which the server partly skipped for repeated prompts (latency without it gives the same ratio). Because Qwen3 always reasons when thinking is enabled, the main study’s measure of how often a model reasons is uninformative here; the measure is the number of reasoning tokens.

Pre-registration. An addendum to the pre-registration, written after a plumbing pilot on the two smoke-test tasks and frozen with SHA-256 hashes before the first generation on an evaluation task, made the reasoning shift observed for the Claude models the primary, confirmatory hypothesis (Q1: more reasoning tokens for Spec, one-sided). It repeated the main study’s hypotheses: shorter answers (Q2a), fewer output tokens and lower latency (Q2b, Q2c; H1), higher validity (Q3; H2) and a difference in correctness (Q4; H3), with the same task-level exact tests and Holm’s correction across the six tests; the runs with thinking disabled form an exploratory ablation. The pilot had shown more reasoning for Spec on both smoke tasks, which the addendum discloses; nothing was changed after it. One deviation occurred: a generated page that hung while loading made the orchestrator raise an exception instead of recording the page as invalid. After a fix, the job was completed from its saved generation rather than generated anew, so that its actual outcome (invalid) was kept.

Results with thinking enabled. Tables 7 and 8 summarise the results. The reasoning shift did not replicate: Qwen3-14B spent a median of 3,370 reasoning tokens on HTML/JS and 3,686 on specifications (median per-task ratio 1.07 [0.95, 1.33], Holm $p = 0.29$), and specifications drew more reasoning in only 11 of 20 tasks. In an exploratory breakdown, the only clear shift was for algorithm tasks (median ratio 1.57), the domain with the strongest shift for the Claude models, whereas protocol tasks drew less reasoning for specifications (0.76). The answer itself was again much shorter for specifications (ratio 0.52), but it made up at most a third of the output, so that total output (0.87 [0.76, 1.04]) and latency (0.89 [0.77, 1.11]; medians 104 vs 95 s) were not significantly reduced. The specification format was, however, far less reliable than for the Claude models: 13% of first attempts and 33% of jobs after repair were valid, against 85% and 95% for HTML/JS, and 17% against 62% of jobs passed all semantic tests (Holm $p = 0.004$);

Table 7: Qwen3-14B follow-up: outcomes by mode and format (60 jobs each), scored with the corrected harness. Token and latency columns are medians over first attempts; reasoning tokens are counted from the streamed thinking; latency is measured on one local GPU.

Mode	Format	Valid (%)		All tests pass (%)		Output tokens (median)			Latency (s)
		1st	repair	1st	repair	total	answer	reasoning	
Thinking	HTML/JS	85	95	60	62	4,655	1,559	3,370	103.7
	Spec	13	33	8	17	4,513	906	3,686	95.0
No thinking	HTML/JS	83	97	35	40	1,618	1,618	0	29.2
	Spec	10	32	0	3	982	982	0	18.1

Table 8: Qwen3-14B follow-up: pre-registered tests with thinking enabled (one Holm family of six tests) and the exploratory ablation. Effects are median per-task ratios Spec/HTML/JS for tokens and latency and mean differences in percentage points (pp) for rates, with 95% bootstrap CIs; exact Wilcoxon signed-rank tests on 20 paired tasks, one-sided in the stated direction.

Test	Direction	Effect [95% CI]	<i>p</i>	Holm <i>p</i>
Q1 (primary): reasoning tokens	Spec > HTML/JS	1.07 [0.95, 1.33]	0.071	0.286
Q2a: answer tokens	Spec < HTML/JS	0.52 [0.49, 0.61]	<0.001	<0.001
Q2b (H1): total output tokens	Spec < HTML/JS	0.87 [0.76, 1.04]	0.184	0.552
Q2c (H1): generation latency	Spec < HTML/JS	0.89 [0.77, 1.11]	0.324	0.648
Q3 (H2): valid (with repair), pp	Spec > HTML/JS	−62 [−78, −43]	1.000	1.000
Q4 (H3): all tests pass (with repair), pp	two-sided	−45 [−63, −27]	<0.001	0.004
<i>No-thinking ablation (exploratory): gain in all tests pass from thinking, pp</i>				
HTML/JS	two-sided	+22 [+2, +42]	0.098	
Spec	two-sided	+13 [+0, +30]	0.188	
Spec minus HTML/JS	two-sided	−8 [−32, +17]	0.609	

under the same corrected harness, the Claude models’ specifications were valid in 83–90% of first attempts. The failing specifications show error types also seen for the Claude models (table 6), at a much higher rate: an = prefix inside fields that are always expressions accounts for 39 of the 93 Spec artifacts with an error, followed by invalid JSON and constructs from JavaScript or Python that the expression language lacks, such as `Math.floor`, `lambdas`, list comprehensions or `^` for powers. Most failing HTML/JS jobs produced pages that loaded but showed wrong content. Because most specifications were invalid on the first attempt, the token and latency comparisons for this model contrast mostly valid HTML/JS with mostly invalid specifications.

Ablation with thinking disabled. Without thinking, generation was much faster (medians 29 s for HTML/JS and 18 s for specifications), and here the shorter format did pay: answer tokens fell to 0.64 and latency to 0.65 of the HTML/JS value (both $p < 0.001$, exploratory). Specifications, however, were rarely usable: 32% were valid after repair (HTML/JS: 97%) and 3% passed all semantic tests (HTML/JS: 40%); one specification reached the token limit by repeating the same drawing object. Thinking raised the share of jobs passing all tests by 22 percentage points for HTML/JS and by 13 for specifications; the difference between the formats (−8 points [−32, +17], $p = 0.61$) gives no indication that specifications depend more on reasoning than free-form code.

7 Discussion

Why compression did not pay. The efficiency argument for declarative output assumes that the model’s work is proportional to the length of what it writes. For the reasoning models we tested, this assumption did not hold, in two different ways. The Claude models wrote shorter artifacts but reasoned longer before writing them, and for two of three models the added reasoning exceeded the saving. Qwen3-14B reasoned about as much for both formats, but its reasoning was more than twice as long as its answer, so that halving the answer shortened generation only slightly; only with thinking disabled did the shorter format make generation significantly faster, and then specifications were rarely correct. What a reasoning model writes is only part of what it generates, and shortening that part has limited leverage.

Why the Claude models but not Qwen3-14B reasoned more for specifications remains open. It is not simply that Qwen3-14B always reasons: Haiku also reasoned in almost every generation and still roughly doubled its reasoning for specifications. Two mechanisms can produce such a shift, and both may contribute. *Unfamiliarity*: because the format was learned only from the prompt, the model may need to reason about how to use it. *Encoding and planning*: a specification must fix the sequence of states and the expressions that compute them before anything is drawn, in a language without loops inside expressions or user-defined functions, so that iterative computations have to be unrolled

or recast as `while` steps; a program can compute the same values at run time with familiar constructs. The domain pattern is consistent with the second account: for the Claude models the shift is largest for algorithm tasks, whose traces must be encoded step by step, and smallest for physics tasks with closed-form solutions, and algorithm tasks were also the only domain with a clear shift for Qwen3-14B. The two accounts also make different predictions: unfamiliarity predicts that the overhead shrinks as the model becomes familiar with the format, for example with more examples or after fine-tuning, whereas encoding and planning predict that it persists. The Claude interface reported reasoning token counts but not reasoning text; Qwen3-14B’s reasoning text was saved and permits a direct analysis of what the model reasons about, which we leave to future work. Disabling reasoning shows what it buys: for Qwen3-14B it lowered correctness in both formats by similar amounts, so we found no indication that specifications depend more on reasoning than free-form code. For the Claude models this condition remains untested, because no switch of the command-line interface that we tested turned reasoning off.

Implications for declarative generative UI. In our setting, a shorter format learned in context did not make generation significantly faster for any model with reasoning enabled and was cheaper only for Opus on first attempts, and the models did not need it to produce artifacts that run: free-form pages were valid after repair in 95–100% of jobs for every model. A format learned only from the prompt can, moreover, cost reliability: the Claude models produced valid specifications in 83–90% of first attempts, Qwen3-14B in 13%, in line with the observation that small models handle an unseen component catalog poorly [17]. The case for declarative output therefore has to rest on other properties, several of which production systems already exploit. Models can be trained to compose responses from a component library, as reported for Intelligent UI [4], or fine-tuned on a catalog, as shown for A2UI generators [17], which removes unfamiliarity; a static format description can be cached; specifications can be rendered while they stream; and a trusted runtime offers security and consistency that free-form code does not. The first three are hypotheses that the present design can test directly.

Design lessons for model-facing domain-specific languages. Beyond the comparison itself, the errors in section 5.5 suggest three inexpensive changes to such languages: adopt JavaScript semantics wherever a construct looks like JavaScript (dynamic indexing of objects, null-tolerant array functions); avoid syntactic markers whose meaning depends on the field (the `=` prefix, which caused 4 of the 26 Claude errors and was the most frequent error of Qwen3-14B); and surface runtime errors loudly instead of degrading gracefully, since graceful degradation hid five failures from both the repair loop and our own first evaluation.

Scope and limitations. With two model families, 20 tasks and three repetitions per cell, the study resolves large effects such as the shifts in tokens and reasoning, but not differences of 5–10 points in correctness, whose direction is consistent across the Claude models. The open-weight model was a single 4-bit quantised 14B model run locally, so its latencies reflect one consumer GPU, and its token comparisons involve mostly invalid specifications. Besides the output format, the conditions differ in ways that any format comparison of this kind entails, and each can contribute to the reasoning shift: the specification format was learned only from the prompt, the HTML/JS examples contained code for the contract and the controls that could be adapted, the expression language has no loops inside expressions, and all Claude generations used one effort level. Each suggests an ablation that our harness supports without changes to the evaluation: specification prompts with matched demonstrations, HTML/JS that calls a provided runtime library, an extended expression language, further effort levels and model families, and reasoning disabled for the Claude models through the API. The checkers verify the exposed state and its consistency with the rules rather than visual quality or pedagogical value; in the HTML/JS condition the contract is model-written code that could in principle diverge from the drawing, and the harness does not operate the visible controls. We saved mid-timeline screenshots of all 358 artifacts that loaded with a working contract, which would allow a visual rating, although the runtime’s recognisable appearance limits blinding. Because the HTML/JS condition also had to generate its own controls, its output is longer than the explanation alone requires; the token comparison therefore favours the specification, which makes the absence of a clear saving more notable. No condition used streaming rendering. The reference implementations were written by agents of the same model family as the Claude generators, and several failures are strict readings of explicit rules in briefs we wrote; we report them separately (tables 4 and 5).

8 Conclusion

We compared declarative animation specifications with free-form HTML/JavaScript for LLM-generated interactive explanations in controlled experiments with frozen protocols, validated checkers and four models from two families. Neither intuition behind declarative output held for reasoning models. A specification shortened the answer but not generation, because the Claude models reasoned more before writing it and Qwen3-14B’s answer was too small a part of its output to matter; only without reasoning did the shorter format pay off in speed, and then specifications were

rarely correct. A format learned in context was also less reliable than free-form code, markedly so for the smaller open-weight model. Output compression is therefore not the same as generation savings. Future work should test formats that models were trained on, analyse what models reason about when they write a specification, measure streaming and caching, and rate visual quality.

Use of AI assistance

The experimental harness, the reference implementations and parts of this manuscript were developed with the assistance of an AI coding assistant.

References

- [1] OpenAI. New ways to learn math and science in ChatGPT. <https://openai.com/index/new-ways-to-learn-math-and-science-in-chatgpt/>, March 2026. Accessed 2026-10-09.
- [2] Anthropic. Claude builds interactive visuals right in your conversation. <https://claude.com/blog/claude-builds-visuals>, March 2026. Accessed 2026-10-09.
- [3] Google. The Gemini app can now generate interactive simulations and models. <https://blog.google/innovation-and-ai/products/gemini-app/3d-models-charts/>, April 2026. Accessed 2026-10-09.
- [4] OpenAI. GPT-6 and Intelligent UI for everyone. <https://openai.com/index/gpt-6-for-everyone/>, October 2026. Accessed 2026-10-09.
- [5] Yaniv Leviathan, Dani Valevski, Matan Kalman, Danny Lumen, Eyal Segalis, Eyal Molad, Shlomi Pasternak, Vishnu Natchu, Valerie Nygaard, Srinivasan Venkatachary, James Manyika, and Yossi Matias. Generative UI: LLMs are effective UI generators, 2026. URL <https://arxiv.org/abs/2604.09577>.
- [6] Google A2UI Team. Introducing A2UI: An open project for agent-driven interfaces. <https://developers.googleblog.com/introducing-a2ui-an-open-project-for-agent-driven-interfaces/>, December 2025. Google Developers Blog. Accessed 2026-10-09.
- [7] Vercel Labs. json-render: The generative UI framework. <https://github.com/vercel-labs/json-render>, 2026. GitHub repository. Accessed 2026-10-09.
- [8] Thesys. OpenUI: The open standard for generative UI. <https://github.com/thesysdev/openui>, 2026. GitHub repository. Accessed 2026-10-09.
- [9] OpenAI. Latency optimization. <https://developers.openai.com/api/docs/guides/latency-optimization>, 2026. OpenAI API documentation (undated). Accessed 2026-10-09.
- [10] Kunpeng Liao, Yuexiao Ma, Yisheng Lin, Hualin Zeng, Xiwu Zheng, and Rongrong Ji. ALGOGEN: Tool-generated verifiable traces for reliable algorithm visualization. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens, editors, *Findings of the Association for Computational Linguistics: ACL 2026*, pages 3168–3189, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi:10.18653/v1/2026.findings-acl.156. URL <https://aclanthology.org/2026.findings-acl.156/>.
- [11] Mind Lab et al. Macaron-V1: Towards open continual learning with self-improvement and mixture-of-LoRA, 2026. URL <https://arxiv.org/abs/2608.09819>.
- [12] Zhi Rui Tam, Cheng-Kuang Wu, Yi-Lin Tsai, Chieh-Yen Lin, Hung-yi Lee, and Yun-Nung Chen. Let me speak freely? A study on the impact of format restrictions on large language model performance. In Franck Dernoncourt, Daniel Preotiuc-Pietro, and Anastasia Shimorina, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 1218–1236, Miami, Florida, US, November 2024. Association for Computational Linguistics. doi:10.18653/v1/2024.emnlp-industry.91. URL <https://aclanthology.org/2024.emnlp-industry.91/>.
- [13] Will Kurt. Say what you mean: A response to ‘Let Me Speak Freely’. <https://blog.dottxt.ai/say-what-you-mean.html>, November 2024. .txt (dottxt) blog, 2024-11-20. Accessed 2026-10-09.
- [14] Hengxin Fan. Capacity, not format: Rethinking structured reasoning failures, 2026. URL <https://arxiv.org/abs/2606.09410>.
- [15] Qiaosheng Chen, Yang Liu, Lei Li, Kai Chen, Qipeng Guo, Gong Cheng, and Fei Yuan. InteractScience: Programmatic and visually-grounded evaluation of interactive scientific demonstration code generation. In Tong Zhang, Miroslav Dudik, Martin Jaggi, Alekh Agarwal, Sharon Li, Dale Schuurmans, Jerry Zhu, Felix Berkenkamp,

- Hanze Dong, and Alberto Bietti, editors, *Proceedings of the 43rd International Conference on Machine Learning*, volume 306 of *Proceedings of Machine Learning Research*, pages 15585–15613. PMLR, 06–11 Jul 2026. URL <https://proceedings.mlr.press/v306/chen26cf.html>.
- [16] Anthropic. Custom visuals in chat and Cowork. <https://support.claude.com/en/articles/13979539>, April 2026. Claude Help Center. Accessed 2026-10-13.
 - [17] Yingxiang Yang, Weihang Xiao, Ben Bullough, Tushar Deshpande, and Niresh Agarwal. Toward frontier-quality declarative UI generation at small-model cost. In *Proceedings of the 1st International Workshop on Harness Engineering for Generative UI: Everything Evolves, Something Endures*, Harness4GenUI '26, pages 5–12. ACM, October 2026. doi:10.1145/3844177.3844556. URL <http://dx.doi.org/10.1145/3844177.3844556>. arXiv:2609.04184.
 - [18] Google. A2UI v1.0 basic catalog implementation guide. <https://a2ui.org/specification/v1.0-basic-catalog-implementation-guide/>, 2026. Accessed 2026-10-09.
 - [19] Yanzhe Chen, Kevin Qinghong Lin, and Mike Zheng Shou. Code2Video: A code-centric paradigm for educational video creation. In Tong Zhang, Miroslav Dudik, Martin Jaggi, Alekh Agarwal, Sharon Li, Dale Schuurmans, Jerry Zhu, Felix Berkenkamp, Hanze Dong, and Alberto Bietti, editors, *Proceedings of the 43rd International Conference on Machine Learning*, volume 306 of *Proceedings of Machine Learning Research*, pages 15370–15396. PMLR, 06–11 Jul 2026. URL <https://proceedings.mlr.press/v306/chen26bx.html>.
 - [20] Tiffany Tseng, Ruijia Cheng, Andrew M. McNutt, and Jeffrey Nichols. Keyframer: A design probe for exploring LLM assistance in 2D animation design. In *2025 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 33–45. IEEE, 2025. doi:10.1109/vl-hcc65237.2025.00014.
 - [21] Yupeng Xie, Zhenyang Wang, Liangwei Wang, Jiayi Zhu, Zhouan Shen, and Yuyu Luo. DataMagic: Authoring data videos through declarative multi-agent orchestration, 2026. URL <https://arxiv.org/abs/2609.33403>. Accepted at IEEE VIS 2026.
 - [22] Sneha Gathani, Sirui Zeng, Diya Patel, Ryan Rossi, Dan Marshall, Cagatay Demiralp, Steven Drucker, and Zhicheng Liu. Bridging natural language and interactive what-if interfaces via LLM-generated declarative specifications, 2026. URL <https://arxiv.org/abs/2604.07652>.
 - [23] John T. Stasko. Using student-built algorithm animations as learning aids. In *Proceedings of the Twenty-Eighth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '97, pages 25–29. ACM, 1997. doi:10.1145/268084.268091.
 - [24] Younghoon Kim and Jeffrey Heer. Gemini: A grammar and recommender system for animated transitions in statistical graphics. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):485–494, 2021. doi:10.1109/TVCG.2020.3030360.
 - [25] Jonathan Zong, Josh Pollock, Dylan Wootton, and Arvind Satyanarayan. Animated Vega-Lite: Unifying animation with a grammar of interactive graphics. *IEEE Transactions on Visualization and Computer Graphics*, 29(1): 149–159, 2023. doi:10.1109/TVCG.2022.3209369.
 - [26] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 50208–50232. PMLR, 21–27 Jul 2024. URL <https://proceedings.mlr.press/v235/wang24h.html>.
 - [27] Ivan Yee Lee, Loris D’Antoni, and Taylor Berg-Kirkpatrick. The format tax, 2026. URL <https://arxiv.org/abs/2604.03616>.
 - [28] Barbara Tversky, Julie Bauer Morrison, and Mireille Bétrancourt. Animation: can it facilitate? *International Journal of Human-Computer Studies*, 57(4):247–262, October 2002. ISSN 1071-5819. doi:10.1006/ijhc.2002.1017.
 - [29] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945. doi:10.2307/3001968.
 - [30] B. Efron. Bootstrap methods: Another look at the jackknife. *The Annals of Statistics*, 7(1), 1979. doi:10.1214/aos/1176344552.
 - [31] Sture Holm. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, 6:65–70, 1979. URL <https://www.jstor.org/stable/4615733>.
 - [32] An Yang, Anfeng Li, Baosong Yang, et al. Qwen3 technical report, 2025. URL <https://arxiv.org/abs/2505.09388>.

A Additional figures

jobs passing all semantic tests (with repair)

A1_binary_search	3/3	3/3	1/3	3/3	2/3	2/3
A2_lomuto_partition	3/3	2/3	3/3	3/3	3/3	3/3
A3_bfs	3/3	3/3	3/3	2/3	3/3	3/3
A4_hashmap_resize	3/3	3/3	3/3	3/3	3/3	3/3
M1_gradient_descent	3/3	3/3	3/3	3/3	3/3	3/3
M2_backprop_neuron	3/3	3/3	3/3	3/3	3/3	3/3
M3_kmeans	3/3	3/3	2/3	3/3	3/3	3/3
M4_attention	3/3	3/3	3/3	3/3	3/3	3/3
N1_tcp_handshake	3/3	3/3	3/3	3/3	3/3	3/3
N2_tcp_congestion	3/3	3/3	3/3	3/3	3/3	3/3
N3_dns_resolution	3/3	2/3	3/3	1/3	3/3	2/3
N4_go_back_n	3/3	2/3	3/3	1/3	0/3	0/3
P1_projectile	3/3	3/3	3/3	3/3	3/3	3/3
P2_pendulum	3/3	3/3	3/3	3/3	3/3	3/3
P3_damped_spring	3/3	3/3	3/3	3/3	3/3	3/3
P4_rc_charging	3/3	3/3	3/3	3/3	3/3	3/3
S1_monty_hall	3/3	3/3	3/3	3/3	3/3	3/3
S2_dice_sum	3/3	3/3	3/3	3/3	3/3	0/3
S3_fourier_square	3/3	3/3	3/3	3/3	3/3	1/3
S4_bayes_coin	3/3	3/3	3/3	2/3	3/3	3/3
	opus HTML	opus spec	sonnet HTML	sonnet spec	haiku HTML	haiku spec

Figure 4: Jobs passing all semantic tests after repair, per task, model and format (frozen protocol). Darker cells: more of the three repetitions pass.

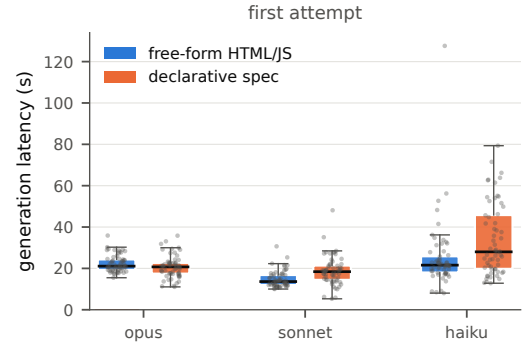


Figure 5: Generation latency of first attempts (boxes: interquartile range; dots: individual jobs). The long upper tail of Haiku’s specification generations coincides with its longer reasoning (fig. 6).

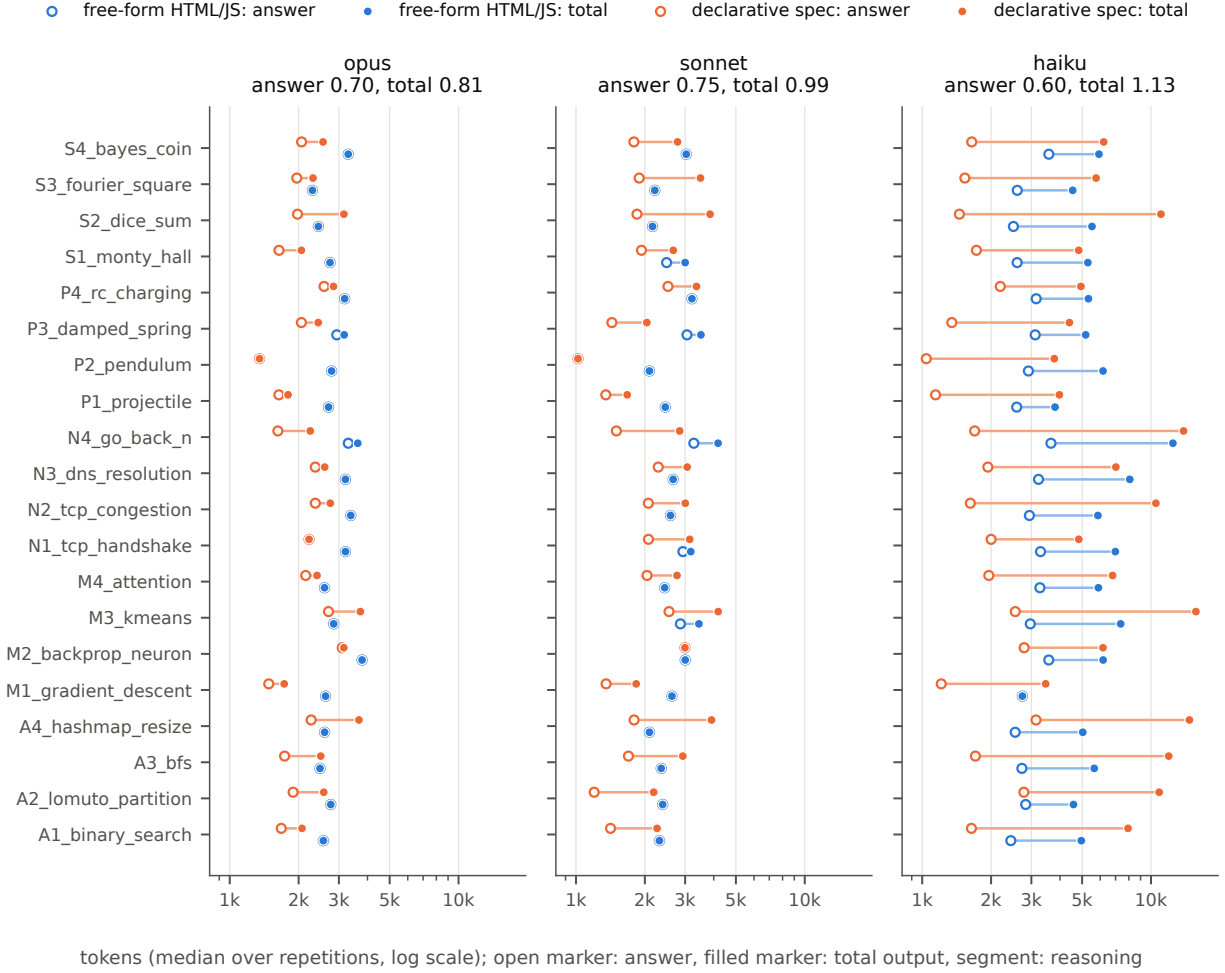


Figure 6: Answer and total output tokens per task (medians over repetitions of first attempts). Each segment runs from the answer tokens (open marker) to the total output (filled marker), so its length is the reasoning; a lone filled marker means no reasoning. Panel titles give the median per-task ratio Spec/HTML/JS for answer and total tokens. Specifications are shorter in answer tokens for almost every task, but with reasoning included they are no longer consistently shorter.